Lesson's Learned in Building a Telerobotic System

Robert D. Steele
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

1 The System

The Supervisory Telerobotics Laboratory (STELER) at the Jet Propulsion Laboratory (JPL) developed a unique local and remote telerobotic system that has been described in earlier articles[1 2 3]. To summarize the system, it is composed of two major subsystems, the Local Site (LS), that provides the ground site including the operator interfaces and the Remote Site (RS) that provides the real time control of the robot and sensors. The software for the RS is known by the acronym MOTES, which stands for Modular Telerobot Task Execution System. Figure 1 pictorial illustrates the two subsystems and their interfaces. In this pictograph the emphasis is on the LS located on the earth, with the RS used in orbit either in a completely autonomously mode or in conjunction with astronauts. The STELER architecture supports any robotic control application where only a minimal bandwidth is available between the local and remote sites.

Figure 2, STELER Context Diagram, shows the relationships between these two components and the Robot. Figure 3, MOTES Functional Diagram,[4] details the functions within MOTES.

The LS was written in C and runs on a combination of SGI workstation and a VME chassis to handle the video capturing portion of the task. The RS was written strictly in Ada and runs in a VME chassis on multiple Heurikon 68020s. The RS is the focus of this article.

1.1 System Goal

As one can image, a remote space environment poses some interesting challenges. The goal of this system was to demonstrate the feasibility of a local-remote architecture for space applications. Many constraints of this environment have been detailed in papers before. The major constraints of the environment include:

- A limited computation environment.
- Time delay in the communications between the Local Site and the on orbit Remote Site of as much as 8 seconds round trip delay.[5]
- Software, including any on-orbit programs for the robot must be flight qualified.
- Any uploaded software must first be flight qualified before the upload to the RS can occur.
- The on orbit system must respond quickly, predictably, and be recoverable to any anomalous situation.

[1]P.G. Backes, M.K. Long, R.D. Steele. Designing Minimal Space for Maximum Space Performance. Proceeding AIAA Aerospace Design Conference, February 3-6, 1992

[2]P.G. Backes, M.K. Long, R.D. Steele. System Architecture for Asynchronous Multi-Processor Robot Control System, Proceeding AIAA Aerospace Design Conference, February 1993

[3]P.G. Backes. Ground-Remote Control For Space Station Telerobotics with Time Delay Proceeding AAS Guidance and Control Conference, February 8-12, 1992

[4]A result of research conducted by Paul G. Backes

[5]R.Aster, J.M. de Pitahaya, and G. Deshpande. Analysis of End-to-End Information System Latency for Space Station Freedom. Technical Report D-8650, Jet Propulsion Laboratory, May 1991.

The large latency in the communication drove much of the design of the STELER system. With such a large latency in the communication, direct control of the robot was not feasible. But the requirements for the real-time control of the robot remain an issue. To meet the real-time control requirements and to deal with the large latency a control scheme was developed that relied on transferring several command blocks that could provide substantial task level control. This scheme is to the paradigm for the control of spacecraft.[6] Instead of transmitting programs to the remote site, a set of data parameters are transmitted that control the execution of the remote site software. This allows the RS software to handle a variety of control modes with no change in the RS software required.

The prototype local and remote site currently communicates via UNIX sockets. The LS sends Task Command Blocks (TCBs) to the RS. The RS sends updates to the LS via Reports. The RS receives TCBs asynchronously from the LS. Reports are transmitted by the remote site at both periodic and aperiodic rates. These TCBs provide the basis for the control of the RS. Parameters within the TCBs specify completely the total execution of MOTES. In the current implementation the TCBs contain approximately 3000 bytes of data and the Reports contain approximately 1500 bytes of data.

This scheme also met the requirement to provide extensive internal monitoring. Besides control parameters, reflex actions were also specified through these data sets. Experiments in the STELER laboratory were performed and tasks such as a docking experiment[7] were completed autonomously using this data block command structure. Superimposed on top of the command tasks, the reflex actions handled anomalous situations, such as excessive forces detected during the docking experiment. Because of the large forces required to complete the docking experiment, the RS software had to respond quickly to any excessive forces detected during the operation. If the RS subsystem did not respond quickly to any anomalous conditions there was a distinct possibility of the failure of the robot.

The data based command structure of the system allows the system to execute a series of commands in a supervised autonomous fashion. If any anomalous action is detected while executing a series of commands, the system reflex mode is initiated and the system moves into a defined safe mode of operation. Dependent upon the task progress this mode could vary from a simple halt of the robot, to a relaxation of forces followed by a halt operation.

2 Ada Software Staffing Profile

The software design and implementation of MOTES was complete with only two full time software engineers assigned to the task, sharing an office, working under one task manager. During one summer two graduate students were assigned to the project

Except the author, none of the personnel working on this project had any Ada experience.[8] The author was initially assigned with the task to teach Ada in one month short course. The author continued with the project serving in a role as both Ada consultant and software engineer. Due in part to the small team size, and having an Ada consultant available, the lack of Ada experience was not a handicap during the project.

3 Rational for the Choice of Ada

The fundamental criteria for selecting Ada for this project was based on the view that all newly developed software for any flight system for Space Station Freedom would be done using the Ada programming language. It was decided that a realistic test platform for a potential flight system should be completed using Ada. Results of the use of Ada for this project could be used to decide if there were any elements of the Ada language that would hinder its use in

[6] Olen Adams of JPL provided insight into this portion of the problem.

[7] W. Zimmerman, P. Backes, R. Steele, M. Long, B. Bon, and J. Beahan. Telerobot Local-Remote Control Architecture for Space Flight Program Applications. Proceeding AAS Guidance and Control Conference. February 1993

[8] Due to a traffic accident, the one engineer with Ada experience was unavailable for the first year of the task.

real time robotic systems. The project then selected a vendor that could supply an Ada compiler that would meet its requirements.

Project requirements were:

1. Only a minimum amount of the budget for the project could be used to acquire both a target compiler and a native compiler.
2. Development would take place on s Sun 4 computer.
3. The target hardware would be Heurikon 68020's.
4. The VME bus would be used as the backplane.
5. The robot to be controlled would be a 7 DOF[9] RRC[10] 1207. The control interface was to be over a nil-3 interface module.
6. It was desirable, but not necessary, to build the software on top of the Windriver vxWorks Operating System.[11]

The bulk of these requirements were levied in a desire to minimize the cost of developing the system. The task had at its disposal Sun 4 and Sun 3 workstations, approximately 12 Heurikon 68020's, several VME chassis, an adequate supply of support cards for a VME chassis, and an RRC[12] 1207 robot and controller were available. By using this equipment, new capital expenditures were minimized. Of course, this reduced the choices for compilers and other software tools.

Part of the selection process included establishing the approximate cost of a development environment and an Ada compiler. The obtained estimates placed the acquisition of both a computer and compiler too expensive for the project to absorb. Some of the Ada compilers evaluated were priced in range of $50,000 to $100,000. This price range was completely out of scope for the size of the task. Many compilers were thus excluded from our evaluation process based strictly on the cost of the products.

The pricing issues continues to discourage the use of Ada. For example, on one current task, the GNU[13] C++ compiler was found to be sufficient for virtually no cost except a donation to the Open Software Foundation.

The final selection based on these criteria was to use the Verdix Ada compiler as the cross compiler and the Sun Ada compiler for as the native compiler. The native compiler was used to construct a simulation of the final system. This simulation was an extremely useful aid in the integration phases. Debugging was simpler in the single processor Sun based environment than the mulitprocessor target environment.

3.1 Benefits of the Ada Choice

An implicit part of the research nature of this project was to determine if Ada could be used to build a real-time robotic control system. At the beginning of the project there was a great deal of uncertainty if the current generation of Ada compilers produced code that would prove efficient for the application. The following were benefits of using Ada.

1. Generic packages were used to provide the framework of the major underlying communication scheme.

[9] Degrees of Freedom
[10] Robotics Research Corporation
[11] JPL Section 347 has a long history and an internal knowledge base about this operating system.
[12] Robotics Research Corporation, Columbus Ohio
[13] Not UNIX

2. The Ada tasking model was used directly to describe the asynchronous nature of the problem.

3. The use of overloaded operators simplified the code of the mathematical algorithms.

4. The potential portability problems were isolated within specific Ada packages.

As described later in this article the generic features of the language were used when defining the basic inter-processing communication within MOTES. The physical layout of shared memory was controlled via facilities of the Ada language to guarantee consistency of the use of memory between programs running on different processors.

4 Design Decisions

4.1 Multiprocessor Decision

Because of the computational requirements and the computational limitations of the Heurikon 68020's[14], it was derided early that the system would be required to use multiple numbers of CPU cards. In the final configuration, 7 Heurikon 68020's were used to support the MOTES software.

Although the Ada language does support multiprocessing and interprocessing communication it dots not require this support to be unified across programs executing on separate CPUs. This limitation forced us to design 0111 own multiprocessor communication.

4.2 Interprocessor Communication

The mechanism chosen to implement the inter-processor communication within MOTES was via shared memory. Figure 4, MOTES Multi-Processor Data Flow, illustrates the relationship of the individual processes to the data stored in shared memory. The basic architecture of Shared Memory is that of the hub or a ring with each processor reading and writing data to arid from the hub as shown in Figure 5. The shared memory interface is handled via generic packages that export read and write procedures. These packages are collected and instantiated by a single Ada package and it is within this package that the layout of Shared Memory is defined.

In the current implementation both Last-In-First-Out (1,11'()) queue structures and First-In-First-Out (111''()) queue structures are supported. Further the 1,11'() structure supports both the single writer/multiple reader and the multiple writer/multiple reader cases. For the FIFO queue only single writer and single reader cases are supported.

Part of the decision to use shared memory was our concern for porting the system to different platforms. So rather than depend upon direct CPU to CPU memory accesses it was fell access to off board memory would be supported on a variety of hardware platforms. One of the benefits of using Ada was that the actual implementation details for the interprocessor communication would be hidden within the Ada package bodies. Additional structures can be added to the current design with no impact to the system. So when porting the system to a new platform, the two generic packages for the LIFO structure and the FIFO structure may have to be modified for the change. Thus no change in the interface to the application code is required.

[14] The clock rate on the processors used was 12.5 MHz

4.3 Processor to Process Mapping

Each Ada task is mapped to a separate vxWorks process. To further aid this process we created a concept know as a wrapper program. For each processor a wrapper program was created that contained all of the application code required for this processor. This wrapper program contains the shared memory interface and the required initialization for each of the processors. Thus the application code that makes up the system may be wrapped differently depending upon the number of processors required.

4.4 Simulator Philosophy

The construction of a simulator was very useful. To mimic a multiprocessor environment each CPU was mapped to an Ada task and this provided a slower but accurate model of the target system.

This processing was mapped to a single CPU. In the target was mapped to an Ada task in the simulation environment. A wrapper program was constructed to provide a framework for the execution of these Ada tasks. This wrapper was required by the rules of the language since an Ada task must be contained within an Ada compilation unit. By introducing a time scale constant it was possible to provide an accurate model of the target system. This simulation environment was built and ran on a Sun 4 computer.

The following is an example of the simulator code:

```
with ism; -- Procedure to initialize shared memory
with Shared_Memory_Variable; -- Location of shared memory variables
with x1; -- Encapsulation of the software that runs on Processor 1
with x2?; -- Encapsulation of the software that runs on Processor 2
procedure Sim is
  package SMV renames Shared_Memory_Variable;
  Abort_Processor_x1 : Boolean renames SMV.Abort_Processor_Array(1);
  Abort_Processor_x2 : Boolean renames SMV.Abort_Processor_Array(2);
  task Processor_1 is
    entry Start;
  end Processor_1;
  task Processor_2 is
    entry Start;
  end Processor_2;
begin
  -- Initialize shared memory.
  ism;
  -- Start the two task, each of which simulate one CPU
  Processor-1 .Start;
  Processor-2.Start;
  -- Execute a delay statement, allowing the simulator to
  -- run for one hour of wall clock time.
  loop
    delay 3600.0;
  end loop;
  -- Abort the two tasks.
  Abort_ Processor_x1 := True;
  Abort_Processor_x2 := True;
end Sire;
```

4.5 MOTES Simulation Capability

Included in the MOTES system was a real-time simulation capability. This capability gave the operator the ability to accurate simulate a robotic application task. This simulation mode used the full set of the MOTES software except the actual inter face from the device driver to manipulate the robot. This was accomplished by having the device driver key o{1' the mode of the system.

Criticisms have been levied concerning this system on this point. The major criticism[15] was that the device driver, considered a *low leve* 1 software function, was required t o be aware of an *high le vel* operator function. The rationale for inserting this feature at this low level was to provide the highest level of confidence in the software that would be controlling the robot. The particular robot used weighs in excess of 100 kilograms and is capable of moving at joint velocities in excess of 1000 degrees per second. Thus even with the operator manning a robot kill switches, a software err or could gen erate a command to t he robot cont roll er that would result in breaking t he robot. So by having the maximum amount of control software execute as part of the simulation a high confidence level could be placed when the comma nd s were executed with the actual robot in the control loop.

Given the nature of the problem the solution of putting a high function inside of the low level device driver seem a reasonable engineering trade off. specially consider the robotic safety issues and the need to guarantee the safe operation of the robot.

4.6 Interface Errors Detecte by the Language

Because the Ada language requires complete speci fication of interfaces many design errors are caught before actual debugging takes place. Other la nguages, such as C, do not provide intrinsic debugging aids as Ada dots, Without these internal checks the project could not of been completed within the schedule allocated.

5 Ada Reuse Sources

At the beginning of the task one c of our goals was to reuse as much Ada cod e as possible. Unfortunately, there was virt ually none that matched our choice of data structures. For example, one need of any robotic system is a robust fully debugged robotic n iath library. Due to the lack of a uniform choice of low level data structures in the Ada robotic community there are no s hared rol 01" math libraries. The Linear Algebra Package written by Allan Klumpp of JPL was used as the basis o f the robot math library constructed by Mark Long for the task. The resources available at Mountain Net 's Ada Repository were u sed to obtain a fcw soft ware tools.

Because much of the robotics work is done using C it has been difficult to port the MOTES software to other projects. Again, because their is not a consistent choice in represen tat ions of data structures is has proved to difficult to share anything more substantial than numerical algorithms.

6 Code Structure

The major data types and math function were encapsulated within a few Ada packages that all the software in the system used. This provided at least one bott leneck during the software development cycle. 0111 compilers were running 011 a Sun 4 using a relatively slow disk drive. Complete compilatio ns of the system would take in excess of 45 minutes. In the 2nd year of the development process we acquired a faster disk drive and upgrades to the Ada

[15] David Lim, formerly of JPL brought this criticism to my attention

compiler. Between these two events the compilation time of the complete system was reduced to approximately 15 minutes. Fortunately, the packages that drove these complete compilations were modified infrequently.

This long cycle for a system compilation was frequent on systems 10 years ago, but software engineers have become accustomed to very quick turn around time. The best solution from both an engineering perspective and a management view is to use the latest development engines to reduce this turn around time.

7 Real-Time Debugging

The classic debugging techniques do not often apply to debugging real-time software. One is often more interested in looking at trends of values than with a particular value. For this project we Used a software oscilloscope program. This provided a means to plot values with respect to time as they were changing. The product StethoScope was chosen for use on this project.

As a side note, this product was only provided with a C interface. Complete Ada bindings were written in a matter of two weeks by one of the graduate students. The case of being able to use third party software written in a different language continues to be one of Ada's strengths.

8 Internet Connectivity

When the author started working on this task, he had no experience with the VME bus, Heurikon boards, or the Windriver vxWorks Operating System. By simple doing a little reading on the Internet bulletin boards I was able to obtain a wealth of information and to build a considerable knowledge base in very short period. Without this resource, problems I encountered would have taken many times longer to solve.

9 Summary of Lessons Learned

- The cost of Ada compilers continue to limit production environments.
- Ease of access to Internet e-mail services aids resolution of problems with vendors.
- Ease of access to the Internet readnews bulletin board increases the effective knowledge base for resolving problems.
- Not using software written in other languages increases development and debug time.
- Lack of Ada experienced engineers is not critical to completing a project on schedule.
- Having at least one Ada experienced engineer to act as a mentor is important when working with engineers inexperienced in Ada.
- Third party software to dynamically debug the system. Software such as StethoScope proved invaluable.
- Workstations with sufficient capabilities to provide minimal compilation time.

10 Acknowledgments

The research described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.